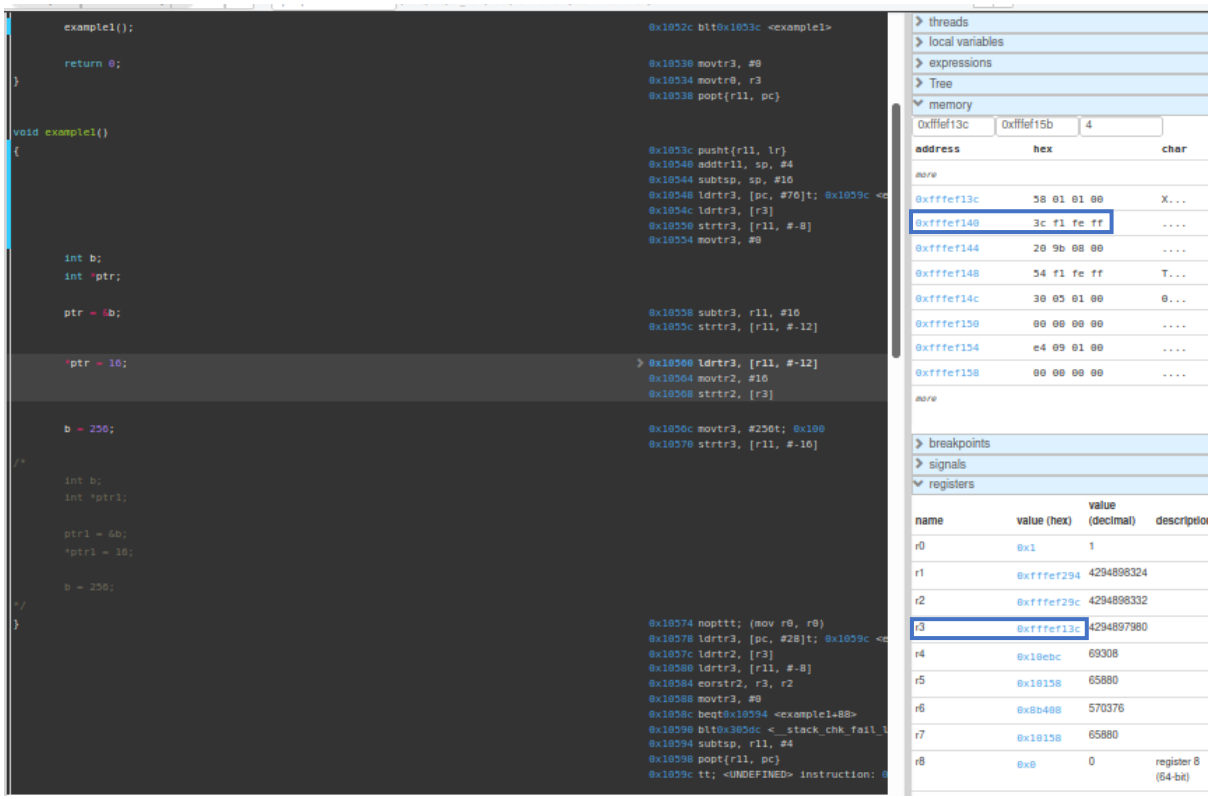
HW6 : lab2 실습

20160394 임효상

※ 설명을 위해 부득이하게 사진을 더 추가, 페이지 수가 약간 더 늘어났습니다.

1) 실습 A

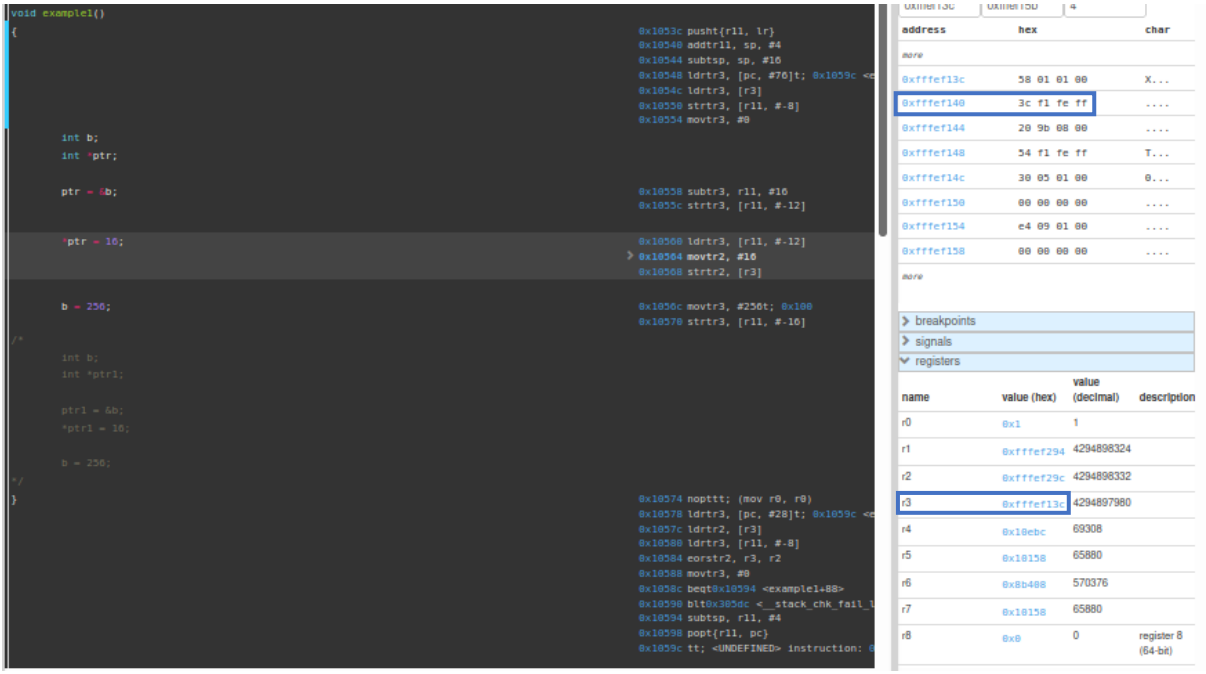


*ptr = 16 명령어를 수행하기 직전의 모습이다.

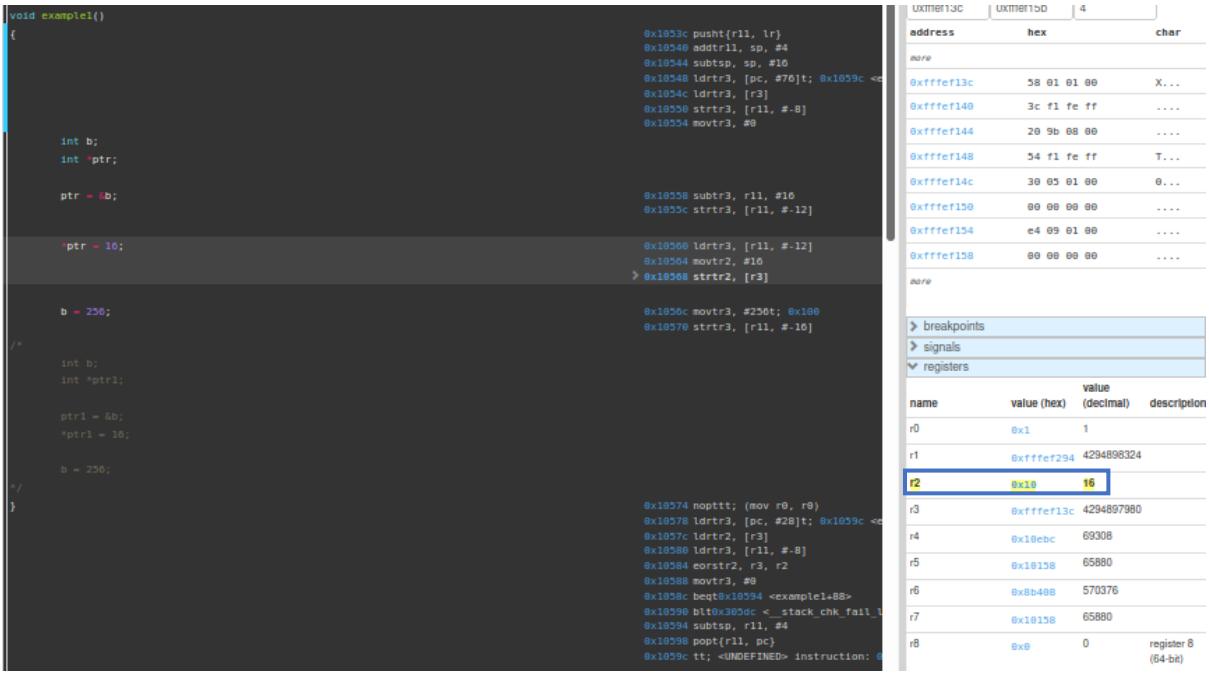
r11에는 sp+4의 주소가 들어있다.

SUB r3, r11, #16 명령어로 인해 r3에는 a의 데이터가 저장된 메모리의 주소(r11-16)가 들어있다.

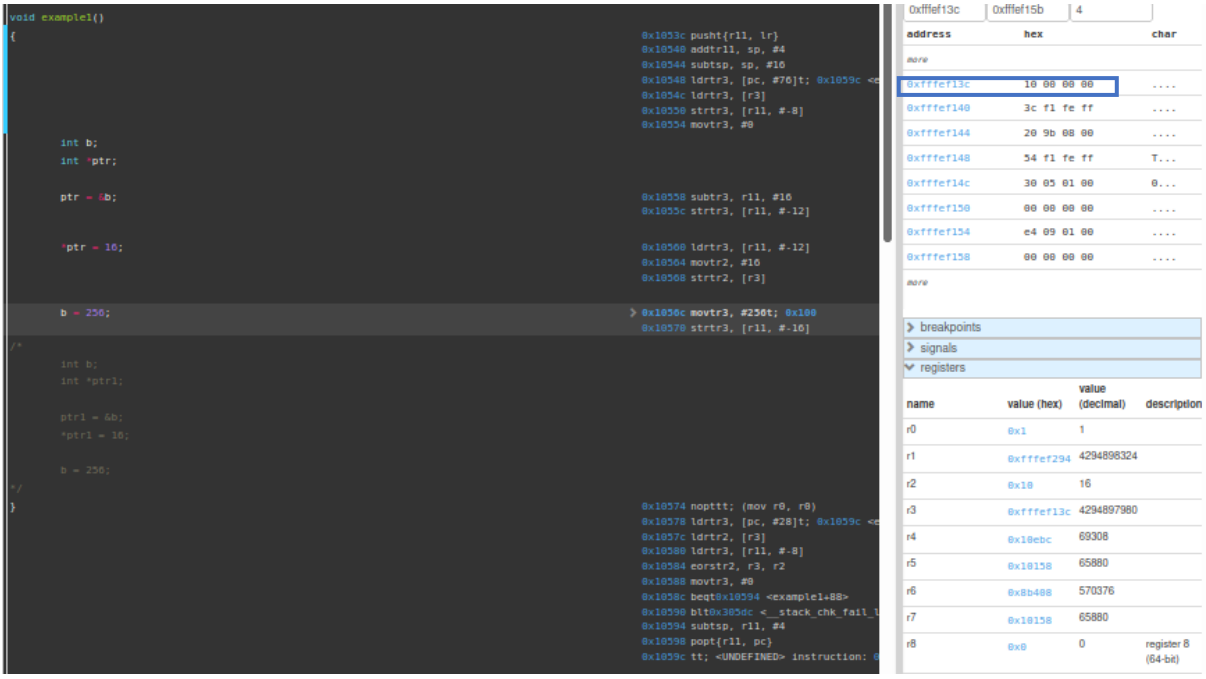
STR r3, [r11, #-12]명령어로 인해 stack의 r11-12번지 주소에는 b의 주솟값이 저장된다.



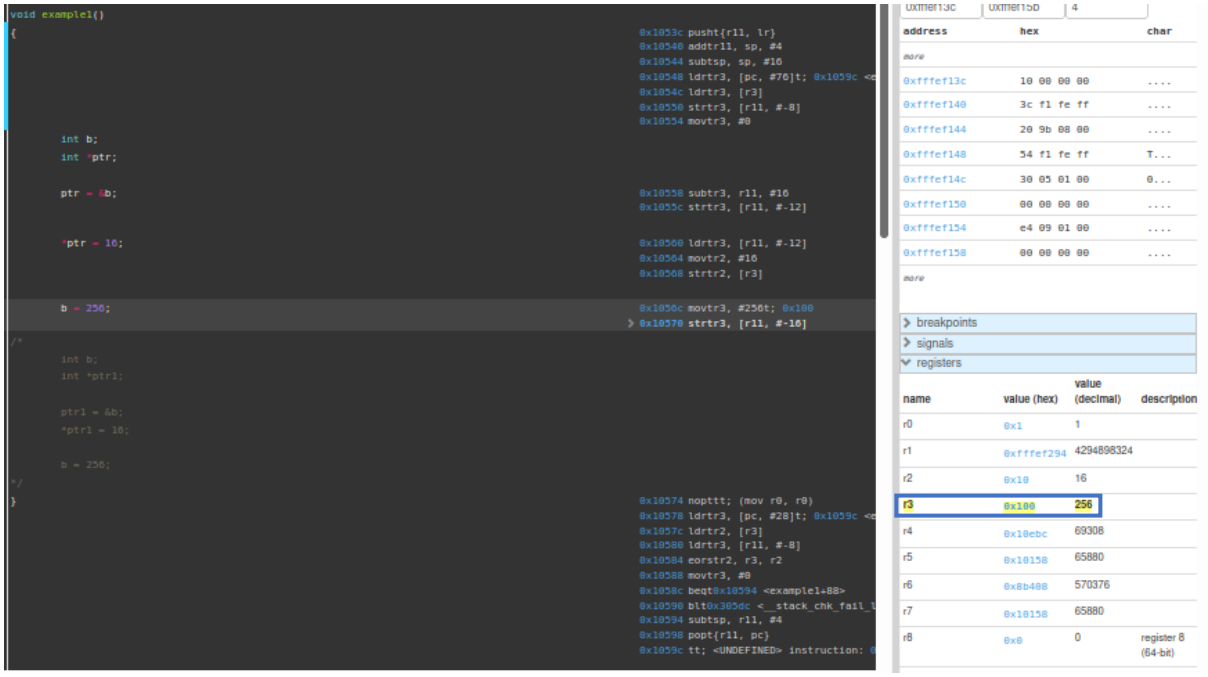
LDR r3, [r11, #-12] 명령어 수행 : r11-12 메모리 주소에는 b의 주솟값(r11-16)이 들어있다. 이 값을 r3에 저장한다. 하지만 기존 r3에 이미 b의 주솟값이 들어있으므로 변화는 없다.



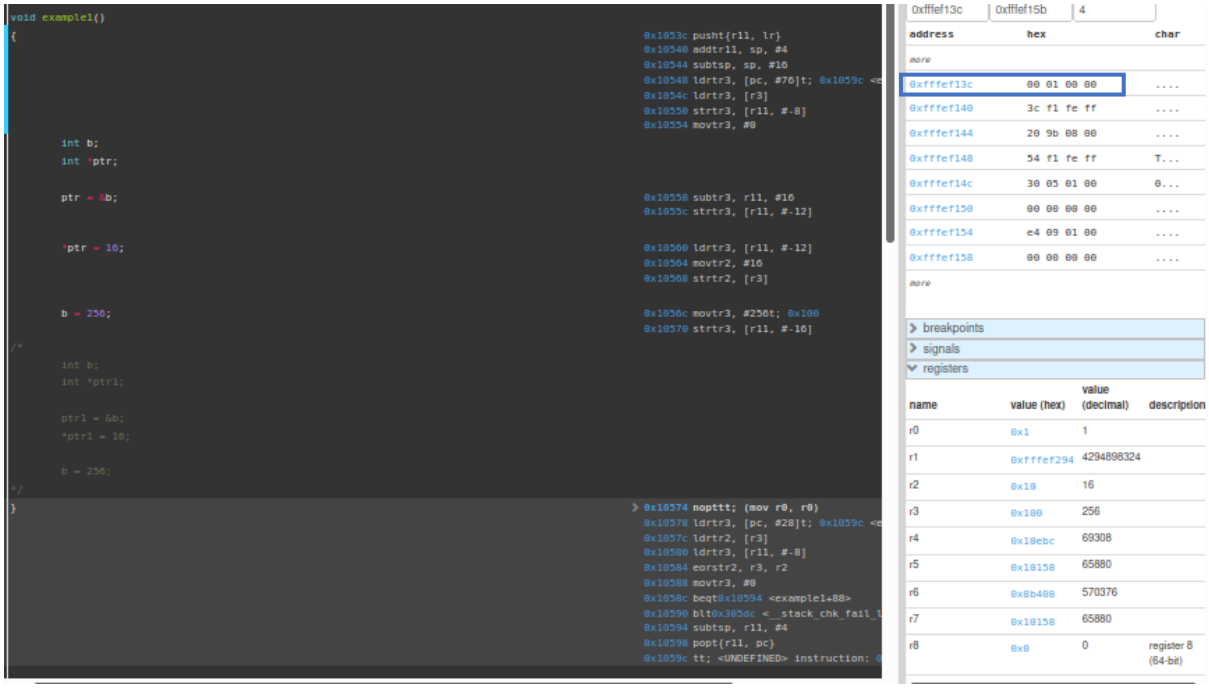
MOV r2, #9 : r2에 16의 값을 저장했다.



STR r2, [r3] : r3에 저장된 값인 b의 주솟값(r11-16)에 r2의 값(16)를 저장했다. 따라서 b의 값은 16다.



MOV r3, #3 : r3에 256의 값을 저장했다.



STR r3, [r11, #-16] : r3에 저장된 값(256)을 r11-16주소의 메모리에 저장한다. 기존 r11-16는 b의 주솟값이므로 b의 값이 256으로 바뀐 것을 알 수 있다.

정리 : 어셈블리에서 포인터를 사용하여 값을 저장하거나 직접 변수에 값을 저장하는 방식은 같은 변수의 주솟값을 사용하여 값을 수정하기 때문에 결과는 동일하다. 하지만 과정에서 약간의 차이가 있다.

변수로 직접 값을 수정하는 경우 레지스터에 값을 임시저장해둔 후, 변수의 주솟값을 직접 찾아가 수정하는 방식이었다.

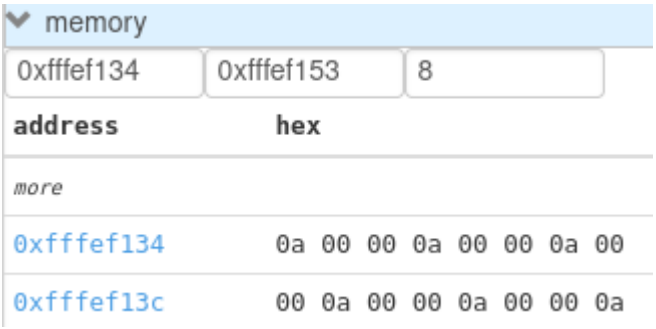
포인터를 사용하여 값을 수정하는 경우 먼저 변수의 주솟값을 레지스터에 저장한다. 이후, 다른 레지스터에 변경할 값을 저장하고, 변수의 주솟값이 저장된 레지스터를 통해 메모리에 접근하여 변수의 값을 수정하게 된다.

2) 실습 B

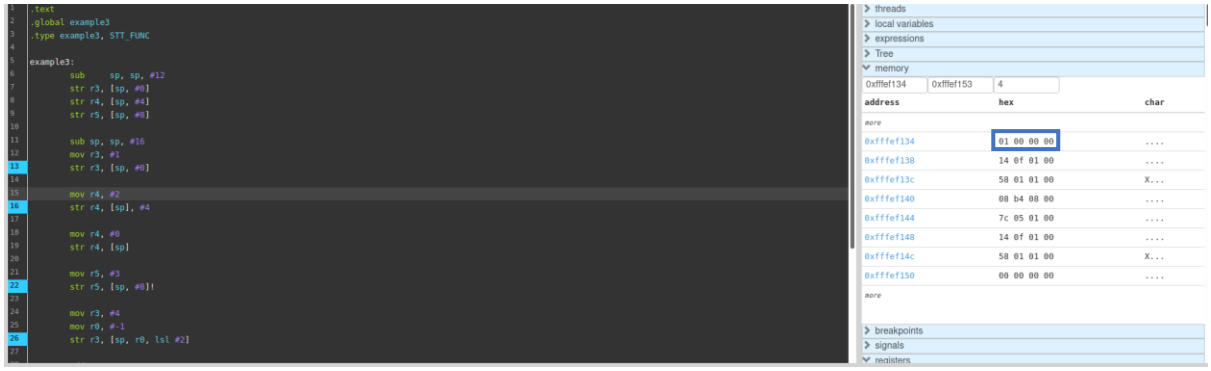
변경된 코드:

```
void example2()
{
        int mem[4] = {0, }; // 16byte array
        char *a;
        int i;

        a = (char*)mem; // Typecasting array's starting address to char-type

        for(i = 0; i < 16; i++)
        {
                if(i % 3 == 0)       // The multiple of 3 of the variable i enters this statement
                        *(a+i) = 10; // *(a+i) becomes 0x0a
        }
}
```

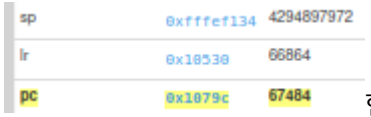
결과화면:



3) 실습 C

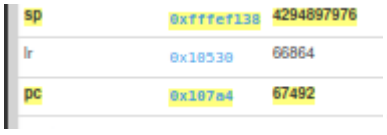


13번째 줄 STR r3, [sp, #0] : Pre-indexing 사용, offset을 먼저 계산하여 r3의 값(1)을 sp+0(0xfffef134)에 해당하는 주소에 저장한다.

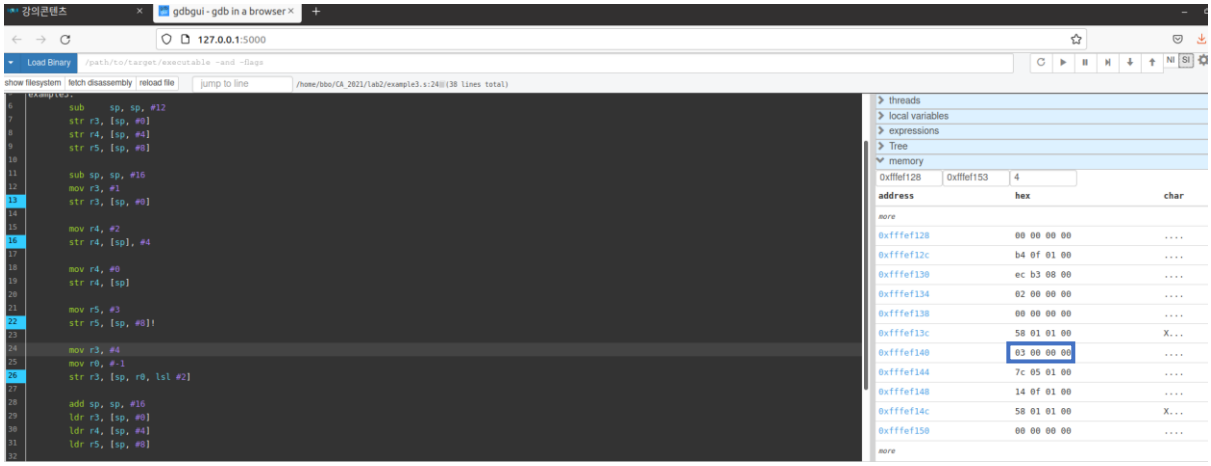


현재 sp값은 0xfffef134이다. 위 코드 수행 후 변화는 없다.

16번째 줄 STR r4, [sp], #4 : Post-indexing 사용, r4의 값(2)를 sp가 가리키는 주소(0xfffef134)에 저장한다. 이후 sp의 값을 4만큼 더한 값으로 업데이트한다.



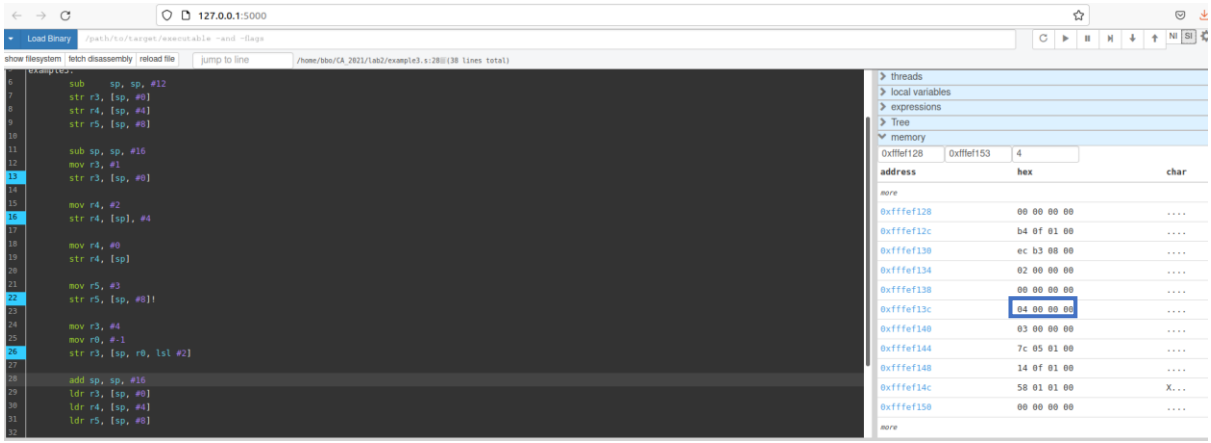
이제 sp의 값은 초기 sp 값에서 +4된 상태다.



22번째 줄 STR r5, [sp, #8]! : Auto-indexing 사용, offset을 먼저 계산하여 r5의 값(8)을 sp+8에 해당하는 주소에 저장한다. 하지만 초기 sp값에서 +4된 상태이므로, 초기 sp값과 비교하면 r5의 값(8)은 sp+12의 위치(0xfffef140)에 저장된다. 이후 sp의 값을 8만큼 더한 값으로 업데이트 한다.

sp	0xfffef140	4294897984
lr	0x18530	66864
pc	0x107b4	67506

이제 sp의 값은 초기 sp의 값에서 +12된 상태다.

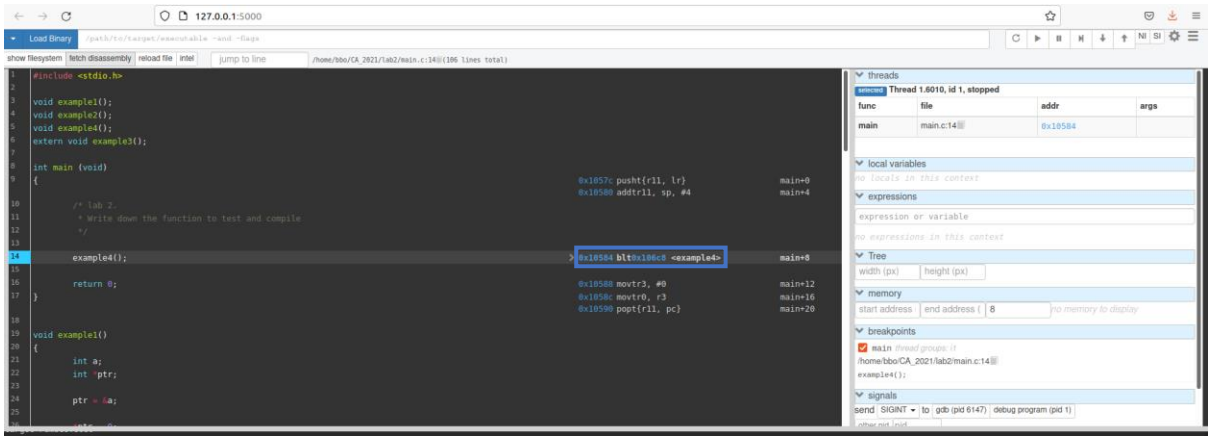


226 줄 STR r3, [sp, r0, lsl #2] : Post-indexing 사용, offset을 먼저 계산하게 된다. r0의 값(-1)을 left shift 2만큼 연산하면 결과값은 -4다. 따라서 r3의 값(4)를 sp-4의 주소에 저장한다. 하지만 초기 sp값에서 +12된 상태이므로, 초기 sp값과 비교하면 r3의 값(4)는 sp+8의 위치(0xfffef13c)에 저장된다.

sp	0xfffef140	4294897984
lr	0x18530	66864
pc	0x107c0	67520

sp 값에는 변화가 없다.

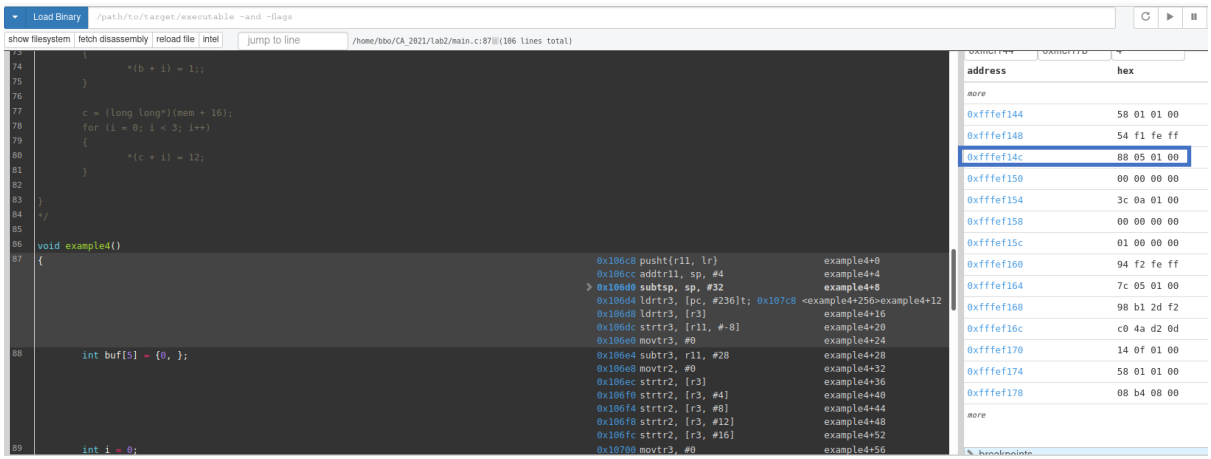
4) 실습 D-1



RETURN을 위해 저장할 PC값 : 0x10584 + 4 = 0x10588

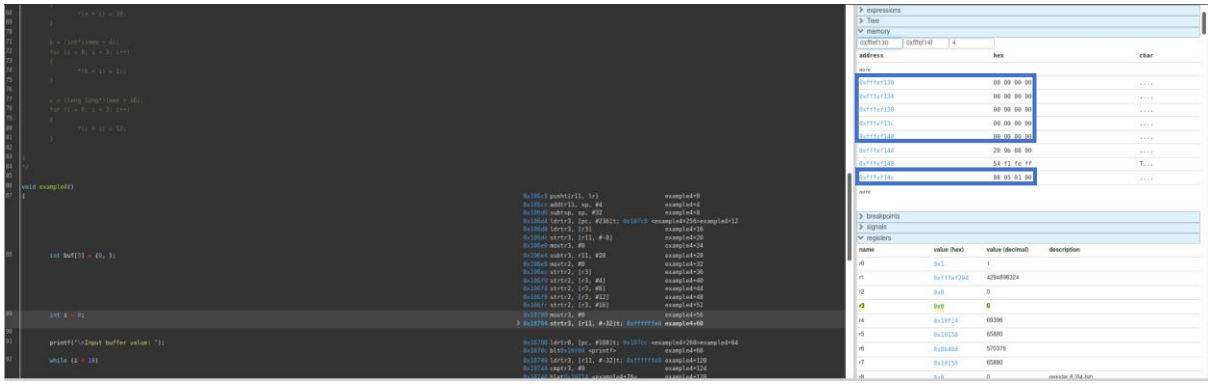
lr	0x10588	66952
pc	0x106c8	67272

branch시 lr에 위의 PC값을 저장, pc는 example4로 이동

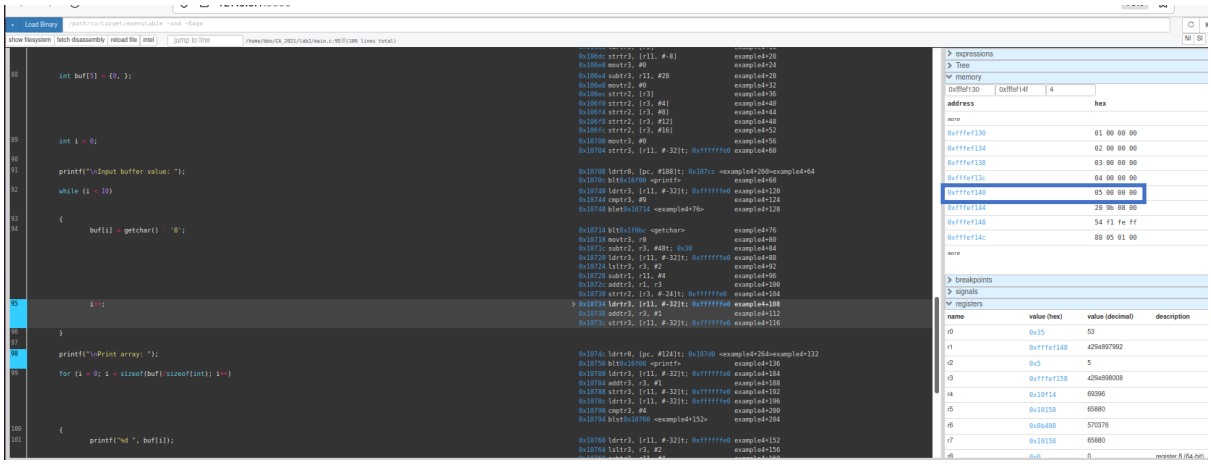


lr을 스택에 push한 주소값 : 0xfffef14c

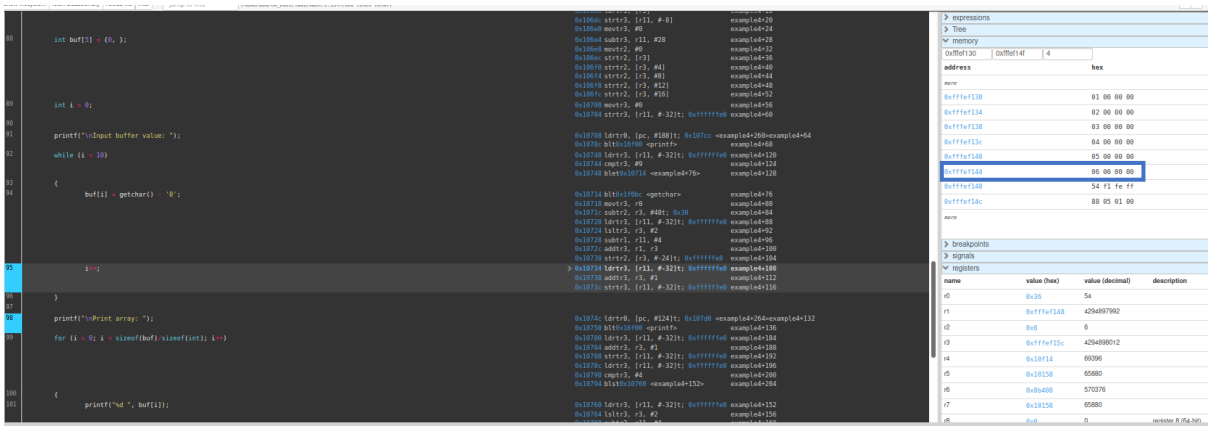
5) 실습 D-2



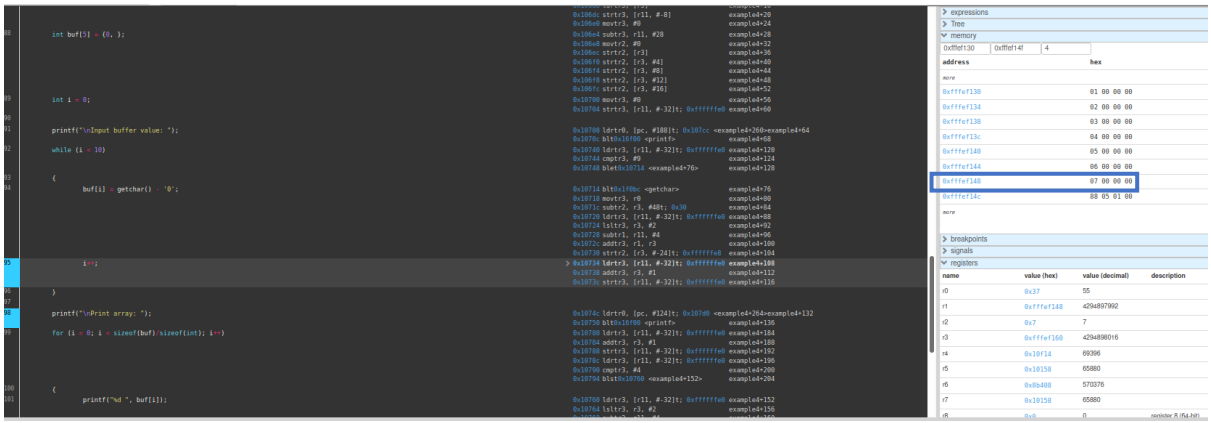
반복문 실행 이전 메모리의 상태를 보면 0xfffef130~0xfffef140까지 배열buf의 공간이고, 0xfffef14c에 return해야할 주소가 있음을 확인할 수 있다. Input으로는 "1234567891"을 주었다.



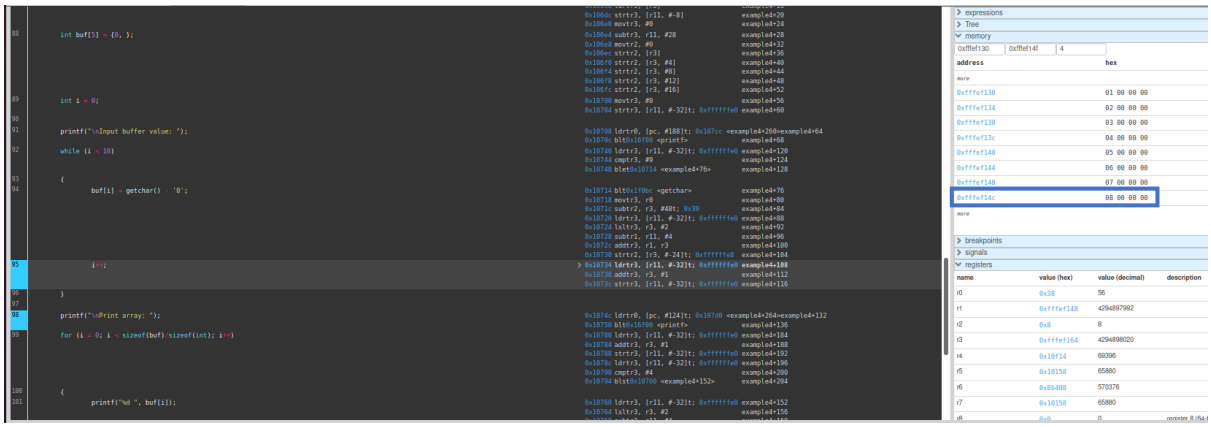
반복문을 5번 실행하여 0xfffef130에서부터 차례로 1,2,3,4,5의 값이 저장되었다. 이제 버퍼가 사용하는 메모리 범위는 모두 사용하였다. 이제부터 배열에 정해진 메모리를 넘어 다른 값들을 변경하게 된다.



반복문을 1회 더 진행시켰다. 0xfffef144에 저장된 0x00089b20의 값을 6으로 변경했다.



반복문을 1회 더 진행시켰다. 0xfffef148에 저장된 0xfffef154의 값을 7로 변경했다.



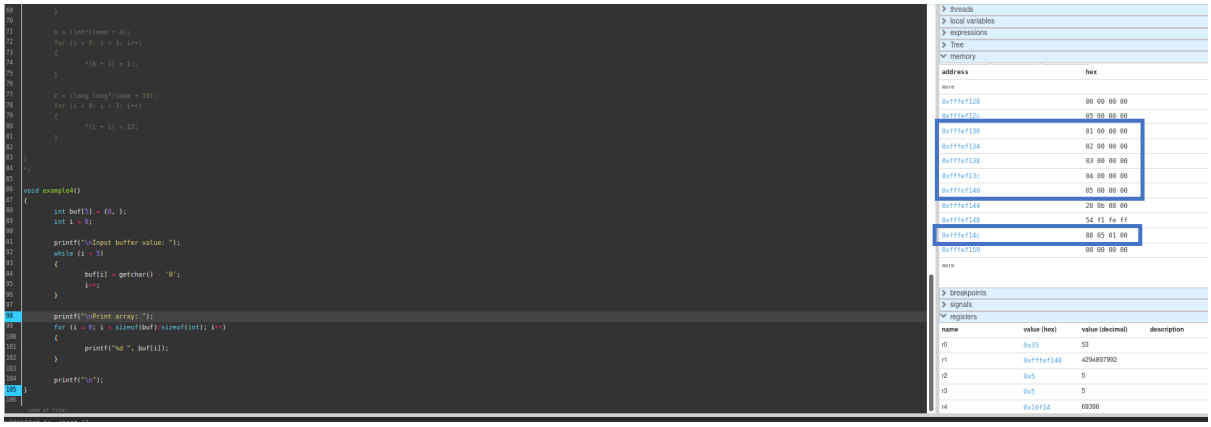
반복문을 1회 더 진행시켰다. 0xfffef14c에 저장된 0x00010588의 값을 8로 변경했다.

0xfffef14c는 return해야할 pc주소가 저장된 메모리공간이다. 이를 8로 변경하여 메모리가 깨진 상황이다.

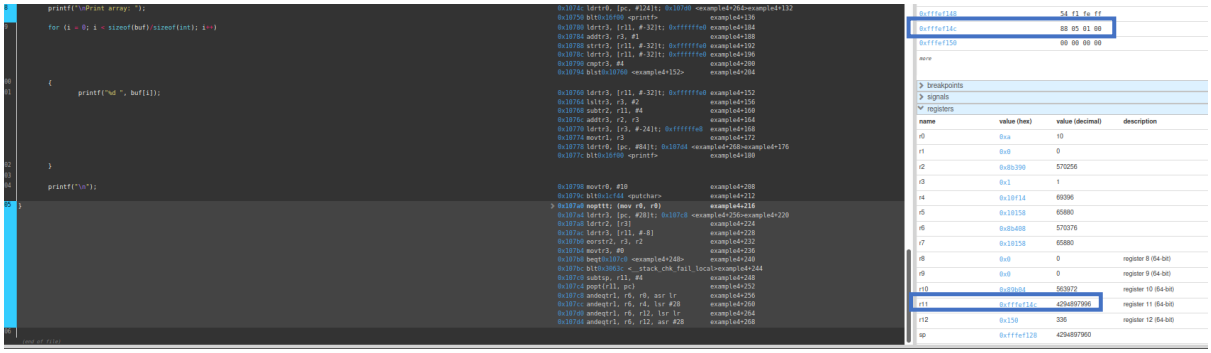
```
bbo@ubuntu:~/CA_2021/lab2$ qemu-arm -g 8080 ./lab2

Input buffer value: 1234567891

Print array: 1 2 3 4 5
*** stack smashing detected ***: terminated
```

example4함수가 끝나고 return할 때 에러가 나오는 것을 확인할 수 있다.

6) 실습 D-3



i를 5로 수정한 후 코드의 모습. i가 5로 buf가 가지고 있는 메모리공간에 딱 맞게 Input을 저장하게 된다. 따라서 buf에 Input을 채워넣는 반복문이 끝나고도 0xfffef14c에 저장되어있는 return 주소 0x10588이 저장되어 있는 것을 확인할 수 있다.



레지스터 11번 r11이 return시 필요한 pc주소를 저장하고 있는 메모리의 주소를 저장하고 있는 것을 볼 수 있다. 이로써 실습 D-2와 달리 정상 수행이 가능하다.

```
bbo@ubuntu:~/CA_2021/lab2$ qemu-arm -g 8080 ./lab2

Input buffer value: 1234567891

Print array: 1 2 3 4 5
bbo@ubuntu:~/CA_2021/lab2$
```

터미널에서도 에러없이 정상 수행됨을 확인했다.